SIENAcollege
Computer Science

Computer Science 210
Data Structures
Siena College
Fall 2019

Topic Notes: Java Review: Arrays, Methods, and Classes

We will spend a few days here at the start refamiliarizing you with and possibly going into a bit more depth about some key programming constructs that are fundamental to programming in Java.

Along the way, we will emphasize some important terminology that you will be responsible for knowing on the exams.

Our Example: Reading and Processing Highway Data

At times throughout the semester (and beyond for those going on to take Algorithms), we will be making use of data files and visualization capabilities provided by METAL: Map-Based Educational Tools for Algorithm Learning. There is a link to the project's web site, `http://courses.teresco.org/metal/`, on the course home page. This project has been developed over the course of several years, and was enhanced greatly by students participating in recent Summer Scholars projects.

The basic idea is that the project has a large collection of data files, each of which represents some highways in some part of the world. Those highways are represented by *waypoints* at major intersections, each of which has a unique label and coordinates, and *connections*, each of which knows which two waypoints it connects and the name of the route or routes that form that connection. It's not important to us now, but these can be thought of as graph structures, with waypoints as graph vertices and connections as graph edges.

To start, we will check out what the data looks like in map form, then see the data file that corresponds.

To view in map form, we use METAL's Highway Data Examiner (HDX), which is at `http://courses.teresco.org/metal/hdx/`, then to see the same data in its text file form by looking at the corresponding entry at `http://travelmapping.net/graphs/`. [Specifics omitted – follow along with this in class.]

Our goal will be to find the extreme points: northernmost, southernmost, easternmost, westernmost locations, and longest and shortest labels.

Before we do that, we think about what it means to do a *search* in this context.

- How would a human looking at the data in HDX find the extreme points?
- How would a human looking at the data in HDX find the longest/shortest labels?
- How would a human looking at the data in the TMG file do these?
- And how does a computer need to do this?

We can see this happen on data in HDX by running the "Vertex Extremes Search" on one of the data sets.

Next, we will work together to build a Java program to do a similar search, using a starter skeleton of the code to save some time (you can get yours from the emailed Google Classroom link). Along the way, we will emphasize some terminology and look at additional examples related to the features we are adding.

The starter skeleton includes a class `VertexSearch` with a `main` method that takes the name of a file containing a METAL graph. It then reads about half of that file using a `Scanner`. The TMG files have both vertices and edges, but we will only use the vertices, which we will think of as a field of points.

You should be able to locate examples of each of the following in the starter and know these terms:

- class comment
- import statement
- class header
- class name
- conditional statement
- command-line parameter
- variable declaration
- variable name
- object construction
- primitive type
- object type
- loop
- loop index variable
- loop stopping condition
- loop update
- loop body
- try-catch block

Adding a Custom Class and an Array

We could solve the problem of finding the vertex extremes here by checking each point as it's read in to see if it's further north, south, east, or west of the current northernmost, southernmost, easternmost, or westernmost point so far.

But instead, we will build toward more useful and interesting usages of this data by storing each point as an object of a *custom class*, and keeping all of those objects in an *array*.

As we work through this in class, note the addition of the *constructor*, the *instance variables* (sometimes referred to as *fields*), and the *methods*. With methods, we have *accessor methods* and *mutator methods*. Many specify *formal parameters*, which get values from the *actual parameters* given in each *method call*. Some methods specify a non-`void` *return type*, in which case they must provide a *return value*.

For the array of objects, note the *array declaration*, *array construction*, and the assignment of values to the *array elements* as the result of each *object construction*. In the actual search, we *access array elements*, and loop over the contents of the array.

Additional Array Review

In mathematics, we can refer to large groups of numbers (for example) by attaching subscripts to names. We can represent a set of numbers n_1, n_2,... An array lets us do the same thing with computer languages.

Suppose we wish to have a group of elements all of which have type `KindOfItem` and we wish to call the group `things`. Then we write the declaration of `things` as

```
KindOfItem[] things;
```

The only difference between this and the declaration of a single item of type `KindOfItem` is the occurrence of "`[]`" after the type.

Like all other objects, a group of elements needs to be created:

```
things = new KindOfItem[25];
```

Again, notice the square brackets. The number in parentheses (25) indicates the number of slots to create, each of which can hold one of the elements. We can now refer to individual elements using subscripts. However, in programming languages we cannot easily set the subscripts in a smaller font placed slightly lower than regular type. As a result we use the ubiquitous "`[ ]`" to indicate a subscript. If, as above, we define `things` to have 25 elements, they may be referred to as:

```
things[0], things[1], ..., things[24]
```

We start numbering the subscripts at `0`, and hence the last subscript is one smaller than the total number of elements. Thus in the example above the subscripts go from 0 to 24.

One warning: When we initialize an array as above, we only create slots for all of the elements, we do not necessarily fill the slots with elements. Actually, the default values of the elements of the array are the same as for instance variables of the same type. If `KindOfItem` is an object type, then the initial values of all elements is `null`, while if it is `int`, then the initial values will all be `0`. Thus you will want to be careful to put the appropriate values in the array before using them (especially before sending message to them! – that's a `NullPointerException` waiting to happen).

The following is an example of a Java application that uses arrays of `String`, `double`, and `int`.

See Example: GradeRangeCounter

There are a few items here we haven't used much this semester (the `Scanner`) but which you have seen before. There are also examples of arrays declared and initialized as `final`, and an example of an array of `int` allocated with `new`.

Inserting and Removing with Arrays

We have already seen that there is quite a bit to keep track of when using arrays, especially when objects are being added. We need to manage both the size of the array and the number of items it contains. If it fills, we either need to make sure we do not attempt to add another element, or reconstruct the array with a larger size.

As a wrapup of our initial discussion of arrays, let's consider two more situations and how we need to deal with them: adding a new item in the middle of an array, and removing an item from the end.

For these examples, we will not use graphical objects, just numbers. Arrays can store numbers just as well as they can store references to objects.

Suppose we have an array of `int` large enough to hold 20 numbers.

The array would be declared as an instance variable:

```
private int[] a;
```

along with another instance variable indicating the number of `int`s currently stored in `a`:

```
private int count;
```

and constructed and initialized:

```
a = new int[20];
count = 0;
```

At some point in the program, `count` contains 10, meaning that elements 0 through 9 of `a` contain meaningful values.

Now, suppose we want to add a new item to the array. So far, we have done something like this:

```
a[count] = 17;
count++;
```

This will put a 17 into element 10, and increment the `count` to 11.

But suppose that instead, we want to put the 17 into element 5, and without overwriting any of the data currently in the array. Perhaps the array is maintaining the numbers in order from smallest to largest.

In this case, we'd first need to "move up" all of the elements in positions 5 through 9 to instead be in positions 6 through 10, add the 17 to position 5, and then increment `count`.

If the variable `insertAt` contains the position at which we wish to add a new value, and that new value is in the variable `val`:

```
for (int i=count; i>insertAt; i--) {
   a[i] = a[i-1]
}
a[insertAt] = val;
count++;
```

Now, suppose we would like to remove a value in the middle. Instead of "moving up" values to make space, we need to "move down" the values to fill in the hole that would be left by removing the value.

If the variable `removeAt` contains the index of the value to be removed:

```
for (int i=removeAt+1; i<count; i++) {
  a[i-1] = a[i];
}
count--;
```

The loop is only necessary if we wish to maintain relative order among the remaining items in the array. If that is not important (as is often the case with our graphical objects), we might simply write:

```
a[removeAt] = a[count-1];
count--;
```

In circumstances where we are likely to insert or remove into the middle of an array during its lifetime, it usually makes sense to take advantage of the higher-level functionality of the `ArrayList`, which we will be considering soon.

Here's one more example using arrays developed in class in a previous year:

See Example: CollegeInfo

Additional Methods Review

As programmers, we often find ourselves writing the same or similar code over and over. We learned that we can place code inside of loop constructs to have the same code executed multiple times. There are other situations where we wish to execute some of the same code repeatedly but not all in the same place in our program's execution.

Consider a simple program, which we could have written months ago, that prints the lyrics to *Baa Baa Black Sheep*.

See Example: BaaBaaBad

Notice that we have 4 printouts repeated – these are the refrain of the song. To accomplish this, we either need some copying and pasting or we need to re-type some lines.

A loop wouldn't help us here, as the lines we need to repeat are separated by 4 other lines that do not repeat.

Fortunately, all modern programming languages, including Java, allow us to group sets of statements together into units that can be executed "on demand" by inserting other statements. These constructs go by many names: *functions*, *methods*, *procedures*, *subroutines* or *subprograms*.

Java calls them methods.

Here is our example, using a Java method to group our repeated statements.

See Example: BaaBaaBetter

Our method for printing the refrain looks a lot like the method we know well: `main`. Other than the name (which in this case is "`refrain`") and the fact that we don't have the "`String args[]`" in the parentheses, it has a very similar *method header* to the `main` methods you have seen.

Just like `main`, the *method body* of the `refrain` method consists of a collection of Java statements that will execute when the `refrain` method is called.

We can also see in the `main` method where we *call* the `refrain` method. We simply put its name, followed by `();`

This is somewhat similar to what you have been doing in earlier examples to call methods:

```
System.out.println("Hi!");
keyboard.nextInt();
JOptionPane.showMessageDialog(null,"Hi!");
```

except that there is no name or names before a period before the method name. We can omit it here, because that means we want to call a method in the same class as the method which is making the call.

We could have made our method calls to `refrain` look like the ones we've been using all along:

```
BaaBaaBetter.refrain();
```

Note that we used a method in this case primarily because it allowed us to reduce the amount of repeated code. This is in itself a worthy goal. If we had misspelled one of the words in the refrain, we can change it in one place and it will be corrected in both printings of the refrain.

However, there is another advantage in readability. By having 2 calls to the `refrain` method in our `main` method, it is more clear what we are doing there. With this in mind, we can consider moving parts of our `main` method into a separate method just for clarity.

See Example: BaaBaa2Methods

Passing Parameters to Methods

Some methods work like the ones in the previous examples: they simply perform the same exact task every time they are called.

However, many others will perform functionality that depends on some input. The way we get input to a method is by passing *parameters* (also known as *arguments*) to a method.

We have done this with the methods we have been using all along. What does `System.out.println` print? Whatever we pass as its parameter. What is the range of values returned by a `Random`'s `nextInt` method? It depends on the parameter we pass.

We can pass parameters to methods we write as well.

See Example: NumberInfo

We essentially introduce a variable (which is called a *formal parameter*) to our method that is initialized to whatever value is passed in the parentheses when we call the method (which is called an *actual parameter*).

A slightly more complex example:

See Example: HoursWorked

Here, we pass a `String` parameter to our method. It contains the contents of one line of an input file, and the method is responsible for breaking down that line into its components, which are an employee id number, an employee name, followed by some number of floating point numbers that represent hours worked by day.

This also demonstrates another use of a `Scanner`. If we pass a `String` as the parameter to a `Scanner`'s constructor (recall that we normally pass either `System.in` for keyboard input, or a `File` to read from a file), we can use the `Scanner` methods to read individual words or numbers.

We could alternately move the reading of each line of the file into our method as well.

See Example: HoursWorked2

Now, the parameter to our method needs to be the `Scanner`, so the method will be able to call the `Scanner`'s `nextLine` method.

Passing Multiple Parameters

Nothing stops us from passing multiple parameters to a method or passing information of any data type.

See Example: SumOfSquares

Here, we create a method that accepts two parameters.

Order matters when passing parameters. The first parameter in the call will "match" the first parameter in the method signature, the second with the second, and so on. In this example, we'd get the same result, but that is not the case, in general. Order would matter, for a simple example, if we changed to a "difference of squares" here.

Returning Information from Methods

So far, each method we have written has the same start to its signature:

```
public static void
```

We will now change that last word to go from a "void" method – one which does not return any information, to one which does.

We have used methods that return values all semester, but we have not written any ourselves. Consider some of the following methods:

- `Integer.parseInt`
- `JOptionPane.showInputDialog`
- `nextDouble` of a `Scanner` object
- `nextInt` of a `Random` object

Each of these results in Java performing some task, and giving back to the caller some information.

We can write these kinds of methods as well.

Our first example will be a method that adds up all of the integers between 1 and a given number, and returns the sum.

So a call such as

```
int sum = sumNumbersTo(10);
```

should leave a value of 55 in `sum`.

Such a method and some examples of how to call:

See Example: Sum1ToN

A few quick notes about this example:

- Since our method computes an integer value, we replace `void` in its method signature with `int`. This is the method's *return type*.
- The value we compute that we wish to have our method send back to its caller is specified in a `return` statement. This is the method's *return value*.
- Any code in the method after a `return` statement will not be executed, so is not allowed. An exception might be if we have a `return` inside of a conditional statement (like an `if` or `switch`).

We can see immediately that we have some similar advantages to our `void` methods. The `main` method becomes shorter, and we avoid potentially having to repeat sections of code when we want to compute such a sum in multiple places in our program.

In this case, there is an additional advantage. Some of you may remember that there is a much easier (computationally speaking) algorithm for computing this sum. Rather than looping through all of the numbers and adding each to a running total, we could apply this formula:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

This is a more efficient operation, at least for larger numbers. Here, we do one addition, one multiplication, and one division. (Moreover, the division is a division by 2 - something computers are very good at.)

So if we discover this formula and want to change our program to use it, we need only change our method. We don't need to change anything in `main`!

See Example: Sum1ToNBetter

We next consider an example with a method that takes 4 parameters and returns a `double` value – one to compute the distance between 2 points in the plane.

See Example: Distance

The method itself is not that complicated, but the main program uses it several times, so the example looks more complex than it really is.

See the comments in the code for more.

A Utility Method for Input

We now will revisit a topic from earlier, using methods to provide a better solution. Recall that many of our programs that ask for input have had sections of code that look similar to this:

```
int val;
do {
   System.out.print("Enter a value between 1 and 10: ");
   val = keyboard.nextInt();
   if ((val < 1) || (val > 10)) {
     System.out.println("Value out of range, please try again.");
   }
while ((val < 1) || (val > 10));
```

We can write a method that can accomplish this, and make it generic enough to be useful in a variety of situations.

We will do this by modifying a program that computes a weighted average from a grading breakdown:

See Example: GradingBreakdown

This program works, but as you can see, the `main` method is quite lengthy and includes a significant amount of repeated code. Let's focus on that part of the code that prompts for an reads in category grades and does error checking on those inputs:

```
double labPointsEarned = 0.0;
do {
    System.out.print("How many lab points did you earn (total availa
        LAB_POINTS + ")? ");
    labPointsEarned = keyboard.nextDouble();
    if ((labPointsEarned < 0.0) || (labPointsEarned > LAB_POINTS))
        System.out.println("Response must be in the range 0.0 to " +
    }
} while ((labPointsEarned < 0.0) || (labPointsEarned > LAB_POINTS));
```

There are three items here that differ from one instance of this code block to the next:

- the name of the variable in which to place the result (`labPointsEarned` for this instance)
- the description of the category to be included in the prompt (`"lab"` in this case)
- the upper limit on the range of legal inputs (`LAB_POINTS` in this case)

If we are going to encapsulate this code block in a method, we will need to transfer these bits of information back and forth between the `main` method and the new method.

The description and the upper limit are both known to `main` and will be needed by the new method, so these will become parameters.

The "points earned" we are reading in will be read from the keyboard by the new method, but will be needed back in `main` to accumulate the overall average. This will become a return value.

Finally, our new method will need to know about the keyboard `Scanner` that `main` will still create. So the `Scanner` should also be passed as a parameter.

This gives us the result:

See Example: GradingBreakdownBetter

Additional Custom Classes Details and Examples

You have been writing programs that operate on data (*i.e.*, variables and parameters) that we can categorize in just two ways:

- primitive types, such as `int`, `double`, `char`
- object types, such as `String`, `Scanner`, `Random`, `DecimalFormat`

Here, we focus on these object types a bit more. In particular, we will think more carefully about introducing our own object types into our programs.

Objects and Classes

We already looked at one way to have a single entity in Java refer to multiple items: the array. Arrays are very convenient for many purposes, but they have some important restrictions:

1. all of the items in the array must be of the datatype declared in the array's declaration and construction
2. we can only refer to array elements by their subscript (or index), which must be a number from 0 to $n - 1$ for an array of length n

Among other benefits, *classes* allow us to overcome both of these restrictions.

Every object in a Java program is an entity that can contain both *fields*, or *instance variables* – which are in many ways like the local variables you've been using in your programs almost from the beginning, and *methods* (or *member methods*), which operate on the data in those fields.

The idea of an object is central to the *object-oriented programming* paradigm, which has been very popular since being introduced a few decades ago.

The idea is that we write program components, called classes, which represent templates for the *objects* we wish to represent in our program. For each object, we include fields that are used to represent the state of the object and methods that allow that state to be queried or modified.

A text I used previously includes an example of an alarm clock. They came up with a list of fields that can be used to describe the state of an alarm clock. It is similar to this list:

- the current hour (0-23)
- the current minute (0-59)
- the current second (0-59)
- the alarm hour (0-23)
- the alarm minute (0-59)
- the alarm status (on or off)

And some methods that can be used to modify the state of the alarm clock:

- set current time
- set alarm time
- disable alarm
- enable alarm
- stop currently sounding alarm

We might also consider some methods to query the current state of the alarm:

- get current time
- get alarm time
- get alarm status (on or off)

And then the alarm clock might have some other things it does "on its own" – its state changes as the time proceeds:

- increment time by 1 second
- start sounding alarm

In Java, the functionality of an object is described in a *class*. Beginning programmers in Java need to write classes as containers for our `main`, and in some cases, a few other methods. This is just a small fraction of what a Java class can be used to do.

We look at a mechanism to achieve this by a simpler example. Consider this example, which is written with all code in its `main` method.

See Example: RatiosNoClass

This program maintains information about ratios of integer values. We create two ratios, each represented by 2 `int` variables, and print them, modify them, and compute their decimal equivalents.

But with a class that represents a `Ratio` object, we can *encapsulate* the numbers (the numerator and denominator) into fields, and provide methods to construct (the *constructor(s)*), access (the *accessor methods*), and modify (the *mutator methods*) the fields.

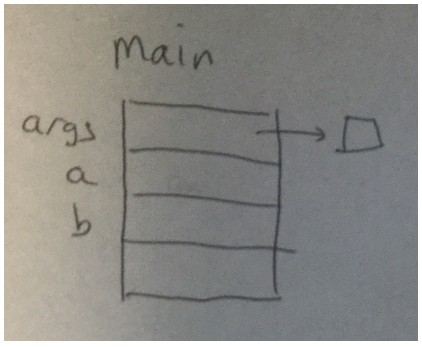
See this example and the extensive comments within for details.

See Example: Ratios

A Memory Diagram

It is important at this point to start thinking carefully about exactly how and where Java allocates memory for the variables in our programs. Throughout the semester, we will make *memory diagrams* of varying detail and complexity. We begin by making one for the example above.

When constructing a memory diagram for a Java application (*i.e.*, a Java program we launch by calling its `main` method), we start by allocating memory for `main`'s parameter and any local variables. We will think more carefully soon about the fact that this memory is allocated on the call stack, but for now, we'll just draw it as a box labeled with the names of any parameters and local variables for our method. In this case:



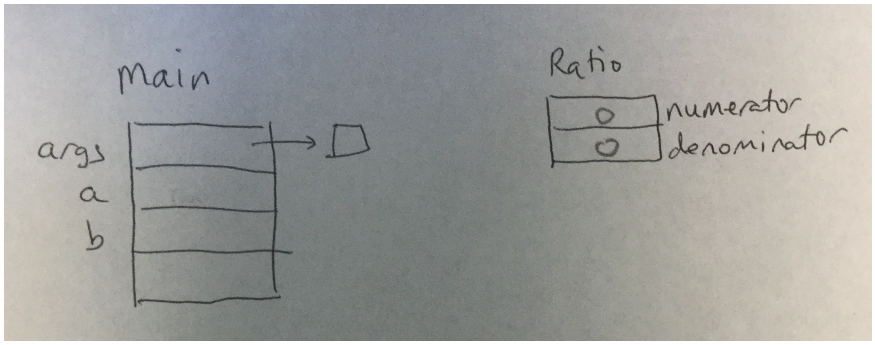
`args` is the one parameter to `main`, and that gets initialized with a reference to any command-line parameters we pass to our program. Since in this case, we aren't expecting any, we will just represent those as the empty box. The `main` method also has two local variables, `a` and `b`, each of which is a reference to a `Ratio` object. At this point in our diagram construction, these have not yet been given values. Java does not provide local variables with any default values, so we leave those boxes blank.

Now, we're ready to look at the actual execution of the `main` method. The first line:

```
Ratio a = new Ratio(4, 6);
```

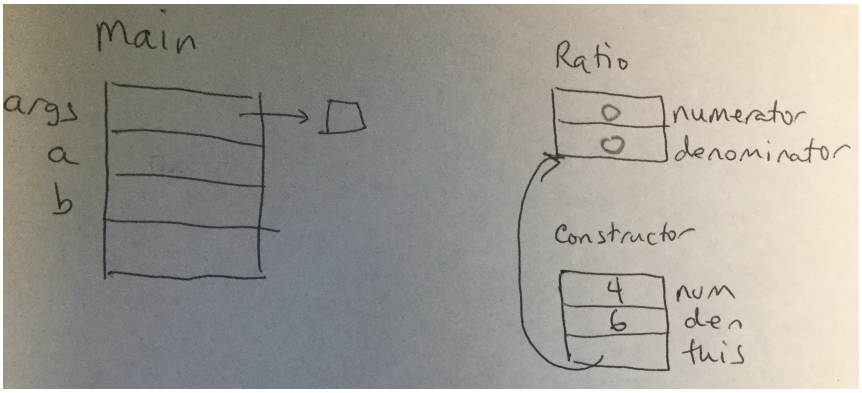
causes many things to occur, and we will consider many of them in some detail, updating our memory diagram for each step that affects it.

That line is an object construction and an assignment of a reference to that new object to a local variable. Before anything can happen, Java needs to find the `Ratio` class, and locate the constructor that takes two `int` parameters. Once it locates the class, it will allocate memory for the instance variables of the `Ratio` class from heap memory.



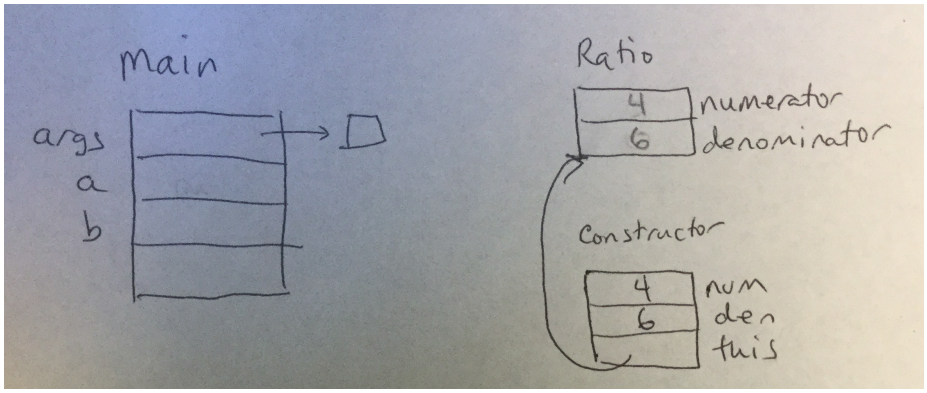
Our `Ratio` class has two instance variables of type `int`, named `numerator` and `denominator`. Java automatically initializes all instance variables with zero, `false`, or `null` values, as appropriate. So our `int` variables get 0.

Next, Java sets up the call to the constructor, which acts much like a method. We get a chunk of memory (this time stack memory) large enough to hold the parameters to the constructor, any local variables in the constructor, and the special `this` reference that will tie this constructor call to its object. The formal parameters `num` and `den` have their values initialized using the actual parameter values from the construction (in this case, 4 and 6). The `this` reference is initialized to point to the instance variables we just created in the previous step.

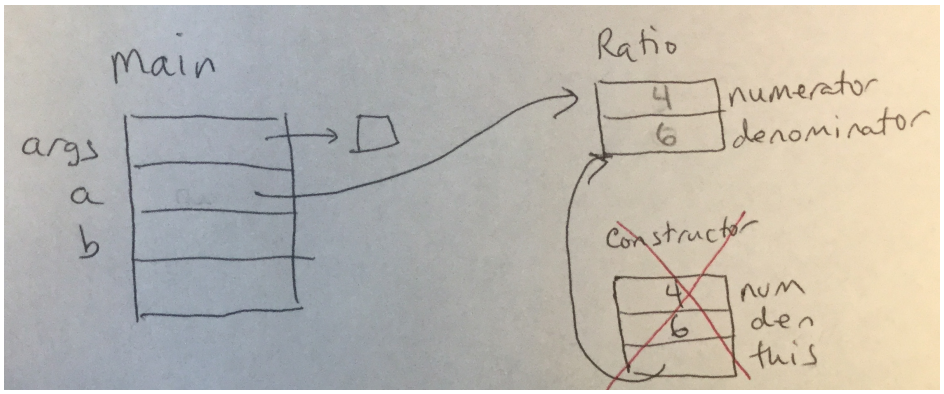


Now we are ready to execute the body of the constructor. This one is pretty straightforward, it's just two assignment statements. But even there, things are not trivial. There are four names involved in the two assignment statements, and Java needs to figure out which of the boxes in our diagram have the values we want to read or are the locations we want to write in each. The process is straightforward. It first looks in its list of parameters and local variables for a matching name. If none is found, it follows its `this` reference to look for a matching instance variable. In our case, `num` and `den` are found in the parameters/locals list in stack memory, and `numerator` and `denominator` are found by following the `this` reference to the instance variable list.

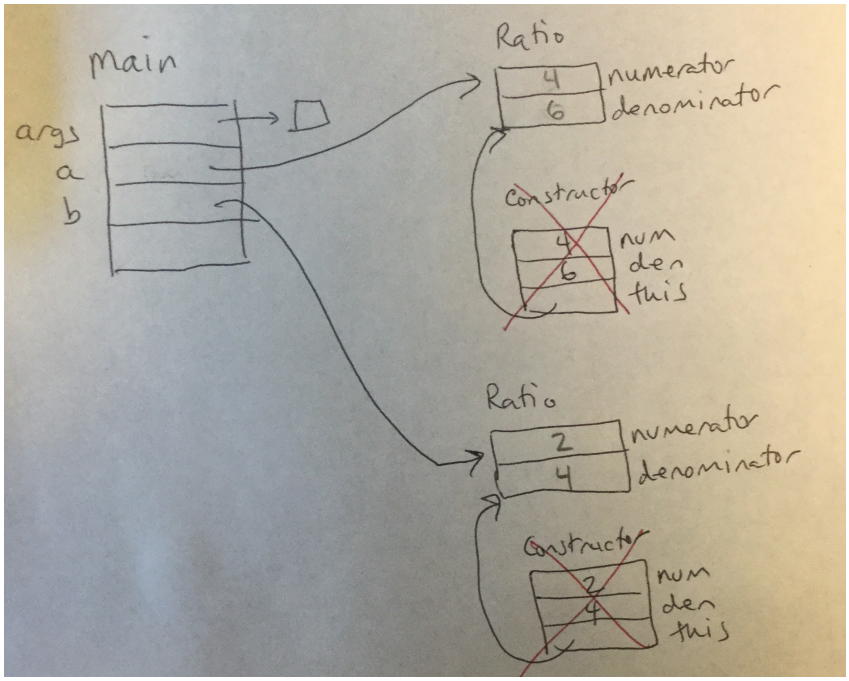
Once these two assignment statements are executed, our instance variables now have taken on the values of the parameters.



When the constructor returns, two things happen: its memory for parameters and local variables goes away, and it returns the reference to the new object's instance variables. In our case, we're storing that reference in `main`'s local variable `a`.



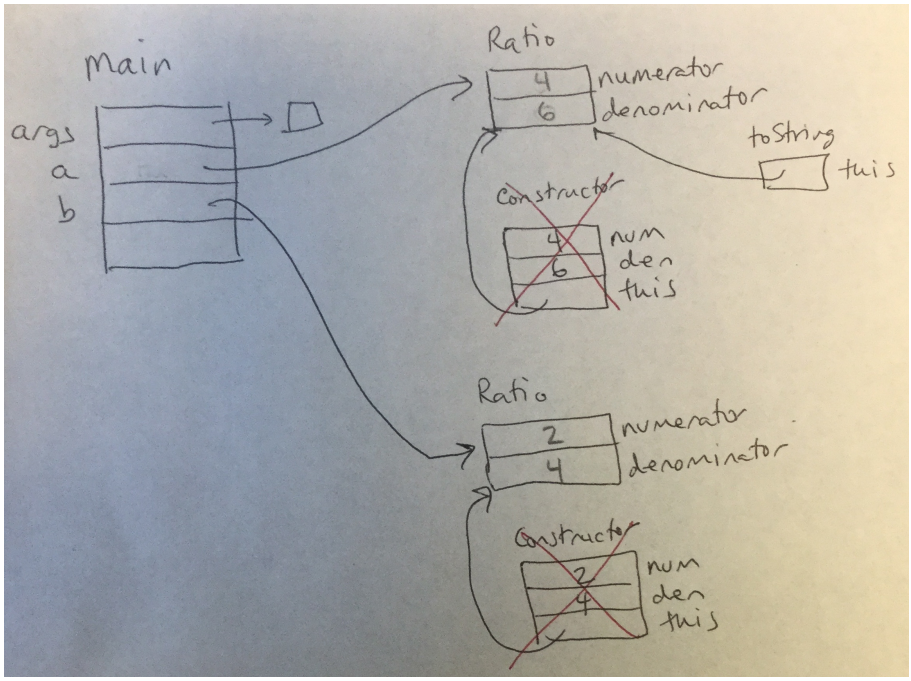
The same set of steps happens when we construct the second `Ratio` and store it in `b`.



Next up in the `main` method is

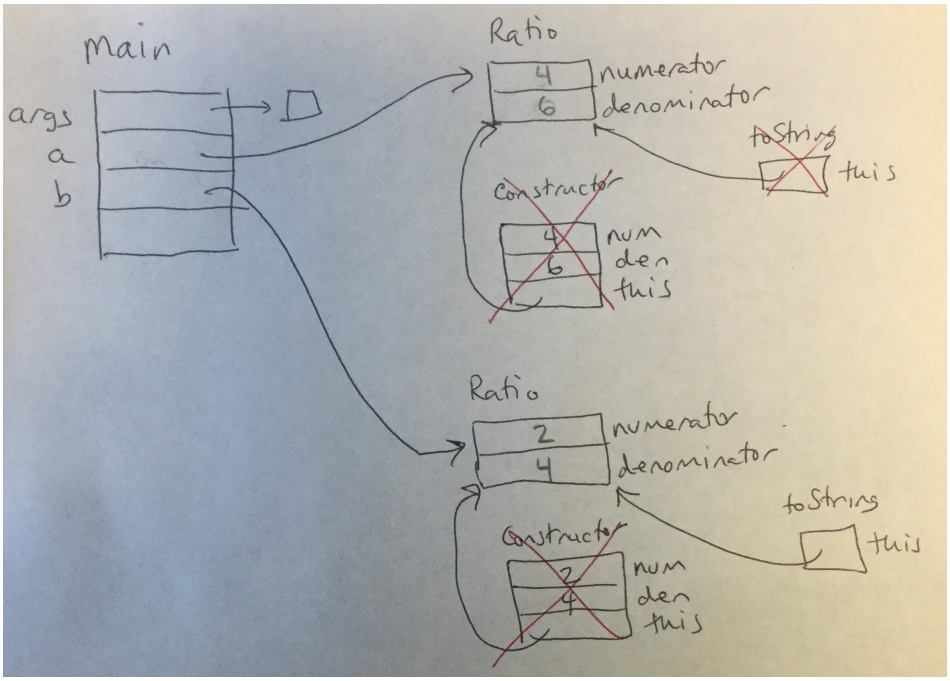
```
System.out.println("Ratio a is " + a);
```

As you might recall, this results in a call to `a`'s `toString` method. Any method call requires Java to allocate memory on the stack for its parameters, local variables, and in the case of non-`static` methods, the `this` reference to the object's instance variables. Since `Ratio`'s `toString` method has no parameters or local variables, it's just `this`.

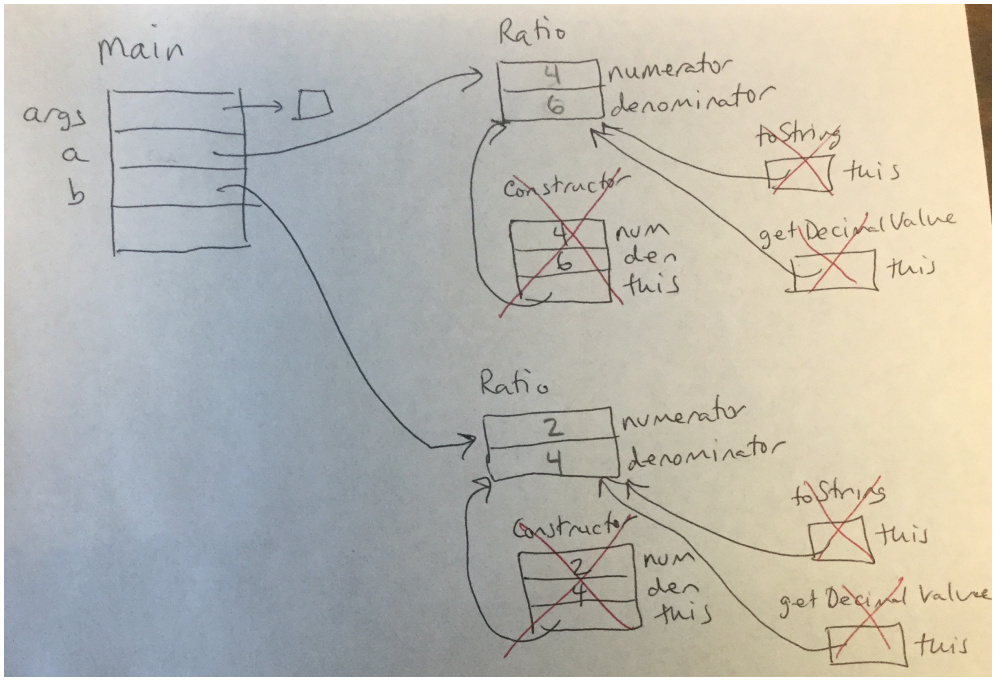


An important thing to notice here is that `toString`'s `this` reference is initialized to the object reference, in this case, `a`.

When the line printing `a` using its `toString` method implicitly completes, its chunk of memory on the stack is deallocated, and we do the same things for the explicit call to `b`'s `toString` method.



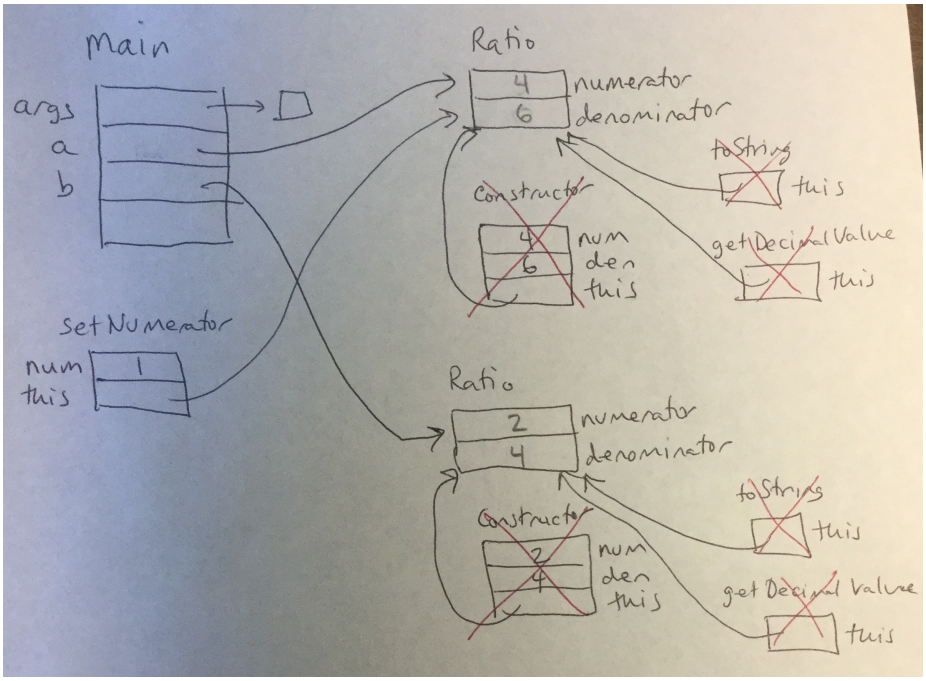
Moving things along a little more quickly, since it's the same idea as what we just saw, the calls to `getDecimalValue` on each of `a` and `b` would also result in chunks of memory on the stack for each of their calls.



A bit more interesting is the call to

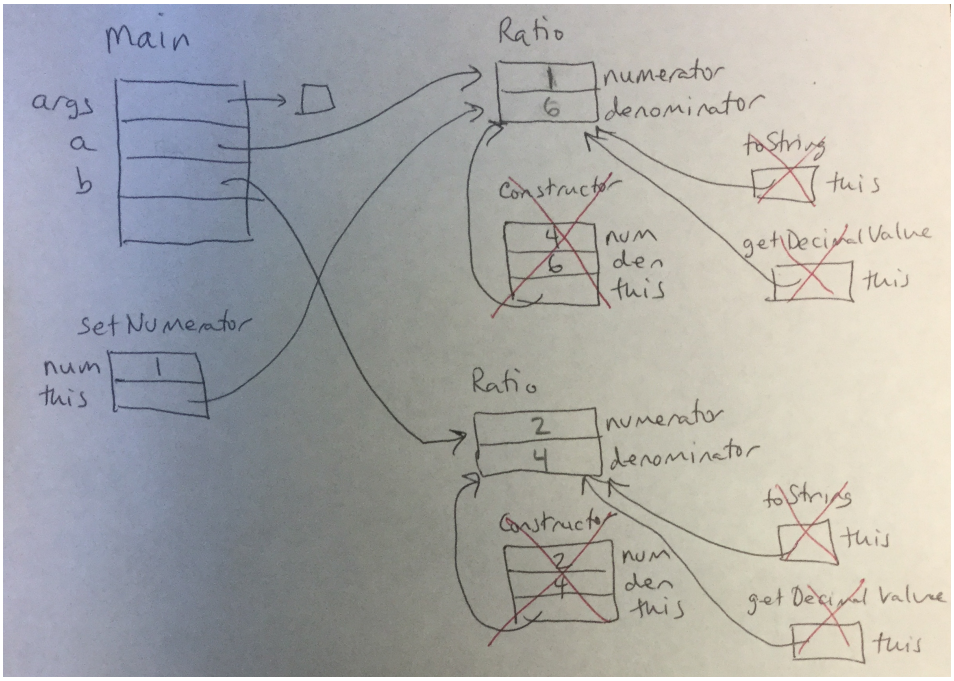
```
a.setNumerator(1);
```

Again, it's a method call, but this one does have a parameter. To set up the method call:



We have a slot for the formal parameter `num`, initialized to the value of the actual parameter, 1. Then the `this` reference, initialized to the object, which is the thing that comes before the `.` in the call.

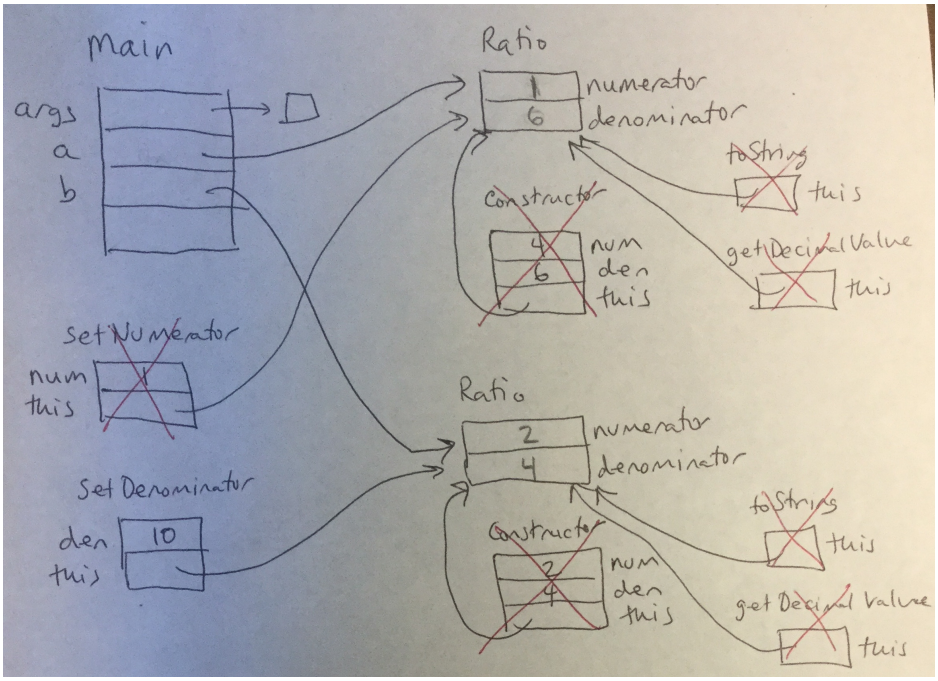
Now the setNumerator method is ready to execute, and it changes the value of `a`'s `numerator`.



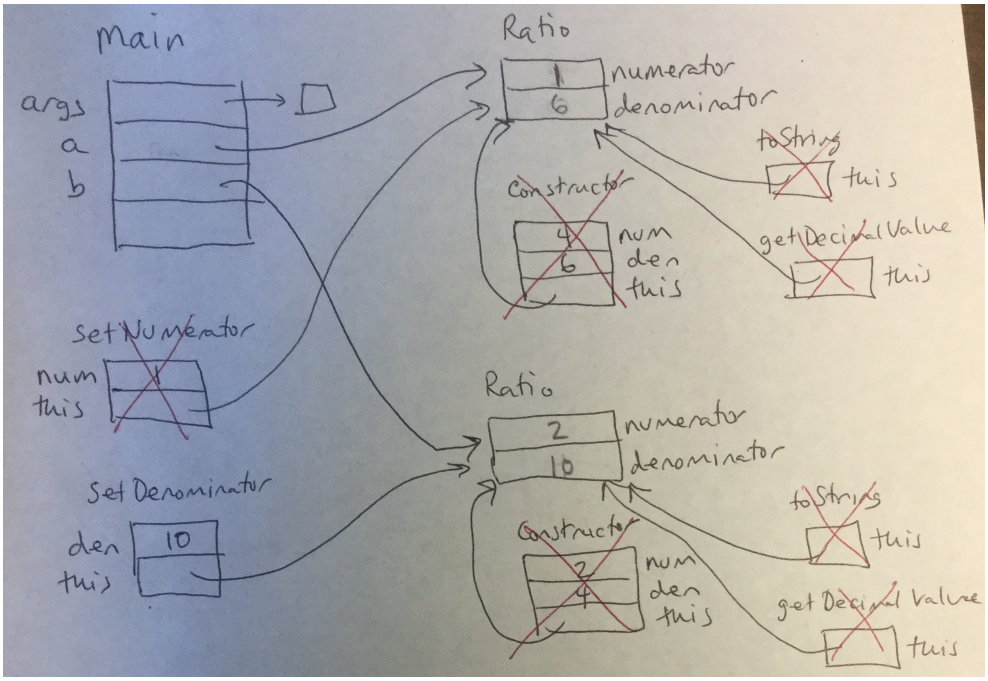
The same process applies to the next call.

```
b.setDenominator(10);
```

The setup:



And when the method executes, `b`'s `denominator` is updated. Shown below is the state of memory just before `setDenominator`.



Object Orientation

The key to successful object-oriented programming is to identify and design the objects in the problem. Looking first at the ratios example, it is essential to notice that the `Ratio` class defines

the data and operations related to a single ratio. It does not know or care how or why the ratios are being used. It simply defines a ratio and methods that operate on one. It is up to other code – in this case the `main` method in the `Ratios` class – to determine when and why `Ratio` objects are instantiated and how they are used.

One popular metaphor for a class definition is to think of it as a cookie cutter. I prefer another: a file cabinet of forms and instructions. Think of the class as a master copy of a form that contains information about an object, how to create one, and instructions for how the object works. Each time we need a new instance of the object, we make a copy of that form and use it to keep track of the details of that object.

As a further example, let's consider a program intended to keep track of a series of items purchased in a store. Each item has a name, a unit price, and a quantity purchased by a customer. Our program will read in a series of these items and track the most expensive, least expensive, the largest quantity purchased, and the largest total cost (unit price times quantity). For simplicity, we will break any ties by the first encountered. So if the most expensive item cost $9.50 but there are two different items each priced at $9.50, we will keep the first and ignore the second.

We will make use of a class that represents one of these items. We will then be able to create an instance of this class for each such item. We can add accessor methods to retrieve the information we need to implement our program.

See Example: PurchaseTracker

One additional item here is the use of a *class variable* in the `PurchasedItem` class. The `DecimalFormat` object we declare and construct with the `static` qualifier is shared among all instances of the class, no matter how many we create.